Part 0
Pre-Intro
Separation Logic

\[ x \rightarrow| y \land y \rightarrow| x \]

\[ x \rightarrow| y \land y \rightarrow| x \]

3
Separation Logic

\[ x|->y \quad * \quad y|-> x \]
Separation Logic

x₁ \rightarrow y
Separation Logic

\[ y \mid \rightarrow x \]
Separation Logic

$x \rightarrow y \ast y \rightarrow x$
Separation Logic

\[ x \mid \rightarrow y \quad \ast \quad y \mid \rightarrow x \]

\[
\begin{array}{|c|}
\hline
x & 10 \\
\hline
y & 42 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|}
\hline
10 & 42 \\
\hline
42 & 10 \\
\hline
\end{array}
\]
Separation Logic

\[ x|->y \ * \ y|->x \]

x=10
y=42

\[
\begin{array}{c}
x \\
42
\end{array}
\]

\[
\begin{array}{c}
10 \\
42
\end{array}
\]
Separation Logic

$x|\rightarrow y$

\[
\begin{array}{c}
x = 10 \\
y = 42
\end{array}
\]
Separation Logic

\[ y \mid \rightarrow x \]

\[ x = 10 \]
\[ y = 42 \]
Separation Logic

\[ x \mid \rightarrow y \quad * \quad y \mid \rightarrow x \]
Part I

Introduction
Example: DisposeTree

- procedure DisposeTree(p)
  local i, j;
  if $p \neq \text{nil}$ then
    $i = p \rightarrow l; j := p \rightarrow r;$
    DisposeTree(i);
    DisposeTree(j);
    dispose(p)
Example: DisposeTree

procedure DisposeTree(p)
local i,j;
if p ≠ nil then
    i = p → l; j := p → r;
    DisposeTree(i);
    DisposeTree(j);
    dispose(p)

An Unhappy Attempt to Specify

\{\text{tree}(p) \land \text{reach}(p, n)\}
DisposeTree(p)
\{\neg \text{allocated}(n)\}
Example: DisposeTree

- procedure DispTree\(\(p\)\)
  local \(i, j;\)
  if \(p \neq \text{nil}\) then
    \(i = p\rightarrow l; j := p\rightarrow r;\)
    DispTree\(i);\)
    DispTree\(j);\)
    dispose\(p)\)

- An Unfortunate Fix

\[
\{\text{tree}(p) \land \text{reach}(p, n) \\
\land \neg \text{reach}(p, m) \land \text{allocated}(m) \land m.f = m' \land \neg \text{allocated}(q)\}\]

DispTree\(p)\)
\[
\{\neg \text{allocated}(n) \\
\land \neg \text{reach}(p, m) \land \text{allocated}(m) \land m.f = m' \land \neg \text{allocated}(q)\}\]
Separation Logic

- In Separation Logic, the spec is just

\[
\{ \text{tree}(p) \} \text{ DispTree}(p) \{ \text{emp} \}
\]

- Key part of proof

\[
\{ \text{p \mapsto } [l\!:i,r\!:j] \ast \text{tree}(i) \ast \text{tree}(j) \} \\
\text{DispTree}(i); \\
\{ \text{p \mapsto } [l\!:i,r\!:j] \ast \text{tree}(j) \} \\
\text{DispTree}(j); \\
\{ \text{p \mapsto } [l\!:i,r\!:j] \} \\
\text{dispose}(p) \\
\{ \text{emp} \}
\]

\[
\frac{\{P\} \text{C}\{Q\}}{\{P \ast R\} \text{C}\{Q \ast R\}} \text{ Frame Rule}
\]
Some Background on Heap Verification

- **Pointer Assertion Logic Engine**
  - Uses MSOL. High complexity, good completeness.
  - (Intentionally) unsound treatment of procedures (framing)
  - No disposal or address arithmetic

- **Boogie.**
  - Sound
  - Improving treatment of frames...
  - Limited induction
  - Class Invariants
  - Relative of ESC
  - No disposal or address arithmetic

- **Sagiv et. al. 3-valued shape analysis**
  - Inferring invariants, good automation
  - Limited treatment of procedures (so far); global, and hard to make local
  - No disposal or address arithmetic
Sep logic has a lot of *locality* built in.

\[
\begin{align*}
\{P\}C\{Q\} \\
\{P \ast R\}C\{Q \ast R\}
\end{align*}
\]

Frame Rule

\[
\begin{align*}
\{P_1\}C_1\{Q_1\} & \quad \{P_2\}C_2\{Q_2\} \\
\{P_1 \ast P_2\}C_1 \parallel C_2\{Q_1 \ast Q_2\}
\end{align*}
\]

Concurrency Rule
Sep logic has a lot of *locality* built in.

\[
\begin{align*}
\{P\} & C \{Q\} \\
\{P \ast R\} & C \{Q \ast R\}
\end{align*}
\]

Frame Rule

\[
\begin{align*}
\{P_1\} & C_1 \{Q_1\} \quad \{P_2\} & C_2 \{Q_2\} \\
\{P_1 \ast P_2\} & \parallel C_1 \quad C_2 \{Q_1 \ast Q_2\}
\end{align*}
\]

Concurrency Rule

Happy with disposal and address arithmetic.
Sep logic has a lot of \textit{locality} built in.

\[
\frac{\{P\}C\{Q\}}{\{P \ast R\}C\{Q \ast R\}} \quad \text{Frame Rule}
\]

\[
\frac{\{P_1\}C_1\{Q_1\} \quad \{P_2\}C_2\{Q_2\}}{\{P_1 \ast P_2\}C_1 \parallel C_2\{Q_1 \ast Q_2\}} \quad \text{Concurrency Rule}
\]

- Happy with disposal and address arithmetic.
- Simple aim: try and see what we can do. So far..
  - Static assertion checking: \textbf{Smallfoot}.
  - Program analysis: \textbf{Space Invader}. 
Part II

Smallfoot Basics
Smallfoot Assertions

A special form\(^1\)

\[(B_1 \land \cdots \land B_n) \land (H_1 \ast \cdots \ast H_m)\]

where

\[
\begin{align*}
H &::= E \mapsto \rho \mid \text{tree}(E) \mid \text{lseg}(E, E) \\
B &::= E = E \mid E \neq E
\end{align*}
\]

\[
E ::= x \mid \text{nil}
\]

\[
\rho ::= f_1 : E_1, \ldots, f_n : E_n
\]

\[
B ::= E = E \mid E \neq E
\]

Smallfoot also has predicates for doubly- and xor-linked lists, but I’ll ignore those.\(^1\)

\(^1\) assertional if-then-else as well
Smallfoot Programs

Procedure declarations

\[ f(\vec{p} ; \vec{v})[P_f] C_f [Q_f] \]

with pre/post and reference params \( \vec{p} \) and value params \( \vec{v} \)

Commands include

\[ x := E \rightarrow f \quad E \rightarrow f := E \quad x := \text{new}() \quad \text{dispose} (E) \]

Loops come with invariants (inferred in Space Invader)
Verification = Symbolic Execution + Entailment Checking

- Inductive Definitions unrolled only on demand (on heap access) during execution.
- Rolled up only after execution, during entailment checking
- The tree definition

\[
\text{tree}(E) \iff \begin{cases} 
\text{if } E = \text{nil} & \text{then emp} \\
\text{else } \exists x, y. (E \mapsto l : x, r : y) \ast \text{tree}(x) \ast \text{tree}(y) 
\end{cases}
\]
Copytree Verification

Just inside the if (where $p \neq \text{nil}$)...

$$\{ p \neq \text{nil} \land \text{tree}(p) \}$$

tree($E$) $\iff$ if $E=\text{nil}$ then emp
else $\exists x, y. (E \mapsto l: x, r: y) \ast \text{tree}(x) \ast \text{tree}(y)$
Copytree Verification

Just inside the if (where \( p \neq \text{nil} \))...

\[ \{ p \neq \text{nil} \land \text{tree}(p) \} \quad \text{unroll it...} \]

\[
\text{tree}(E) \iff \begin{cases} 
\quad \text{if } E = \text{nil} \text{ then emp} \\
\quad \text{else } \exists x, y. (E \mapsto l : x, r : y) \ast \text{tree}(x) \ast \text{tree}(y)
\end{cases}
\]
Copytree Verification

Just inside the if (where $p \neq \text{nil}$)...

\[
\{ p \neq \text{nil} \land \text{tree}(p) \} \quad \text{unroll it...} \\
\{ p \mapsto [l:x, r:y] \ast \text{tree}(x) \ast \text{tree}(y) \}
\]

\[
\text{tree}(E) \iff \begin{cases} 
\text{if } E = \text{nil} \text{ then } \text{emp} \\
\text{else } \exists x, y. \ (E \mapsto l:x, r:y) \ast \text{tree}(x) \ast \text{tree}(y)
\end{cases}
\]
Copytree Verification

Just inside the if (where \( p \neq \text{nil} \))...

\[
\{ p \neq \text{nil} \land \text{tree}(p) \} \quad \text{unroll it...}
\{
\begin{align*}
\{ p & \mapsto [l: x, r: y] \ast \text{tree}(x) \ast \text{tree}(y) \} \\
i & := p \rightarrow l \ ; \ j := p \rightarrow r \\
\{ p & \mapsto [l: i, r: j] \ast \text{tree}(i) \ast \text{tree}(j) \}
\end{align*}
\]

\[
\text{tree}(E) \iff \begin{cases} 
\text{if } E = \text{nil} \text{ then } \text{emp} \\
\text{else } \exists x, y. (E \mapsto l : x, r : y) \ast \text{tree}(x) \ast \text{tree}(y)
\end{cases}
\]
Copytree Verification

Just inside the if (where $p \neq \text{nil}$)...

$$\{p \neq \text{nil} \land \text{tree}(p)\}$$  unroll it...
$$\{p \mapsto [l : x, r : y] \ast \text{tree}(x) \ast \text{tree}(y)\}$$
$$i := p \rightarrow l \;;\; j := p \rightarrow r;$$
$$\{p \mapsto [l : i, r : j] \ast \text{tree}(i) \ast \text{tree}(j)\}$$

\text{tree}\_\text{copy}(ii \;;\; i);\; \text{tree}\_\text{copy}(jj \;;\; j)

s := \text{new}();\; s \rightarrow l := ii \;;\; s \rightarrow r := jj;

\[
\text{tree}(E) \iff \begin{cases} 
\text{if } E = \text{nil} \text{ then } \text{emp} \\
\text{else } \exists x, y. (E \mapsto l : x, r : y) \ast \text{tree}(x) \ast \text{tree}(y)
\end{cases}
\]
Copytree Verification

Just inside the if (where \( p \neq \text{nil} \))...

\[
\{ p \neq \text{nil} \land \text{tree}(p) \} \quad \text{unroll it...}
\{ p \mapsto [l: x, r: y] \ast \text{tree}(x) \ast \text{tree}(y) \}
\]

\[
i := p \rightarrow l; \quad j := p \rightarrow r;
\{
 p \mapsto [l: i, r: j] \ast \text{tree}(i) \ast \text{tree}(j)
\}
\]

\[
\text{tree}_\text{copy}(ii \ ; i); \quad \text{tree}_\text{copy}(jj \ ; j)
\]

\[
s := \text{new}(); \quad s \mapsto l := ii; \quad s \mapsto r := jj;
\{
 p \mapsto [l: i, r: j] \ast \text{tree}(i) \ast \text{tree}(j) \ast s \mapsto [l: ii, r: jj] \ast \text{tree}(ii) \ast \text{tree}(jj)\}
\]

\[
\text{tree}(E) \iff \begin{cases} 
\text{if } E = \text{nil} & \text{then } \text{emp} \\
\text{else } \exists x, y. (E \mapsto l: x, r: y) \ast \text{tree}(x) \ast \text{tree}(y) 
\end{cases}
\]
Copytree Verification

We are left with an entailment

\[ p \mapsto [l:i, r:j] \ast \text{tree}(i) \ast \text{tree}(j) \ast s \mapsto [l:ii, r:jj] \ast \text{tree}(ii) \ast \text{tree}(jj) \]

\[ \vdash \text{tree}(p) \ast \text{tree}(s) \]

\[ \text{tree}(E) \iff \begin{cases} \text{if } E = \text{nil} \text{ then emp} \\ \text{else } \exists x, y. (E \mapsto l:x, r:y) \ast \text{tree}(x) \ast \text{tree}(y) \end{cases} \]
Copytree Verification

We are left with an entailment

\[ p \mapsto [l : i, r : j] \ast \text{tree}(i) \ast \text{tree}(j) \ast s \mapsto [l : ii, r : jj] \ast \text{tree}(ii) \ast \text{tree}(jj) \]

\[ \vdash \text{tree}(p) \ast \text{tree}(s) \]

let me roll it...

\[ \text{tree}(E) \iff \text{if } E = \text{nil} \text{ then emp} \]
\[ \text{else } \exists x, y. (E \mapsto l : x, r : y) \ast \text{tree}(x) \ast \text{tree}(y) \]
Flawed Copytree Failed Verification

When we mistakenly point back into the source tree we are left with an entailment

\[ p \mapsto [l : i, r : j] \ast \text{tree}(i) \ast \text{tree}(j) \ast s \mapsto [l : i, r : j] \ast \text{tree}(ii) \ast \text{tree}(jj) \]

\[ \vdash \text{tree}(p) \ast \text{tree}(s) \]

that we can’t roll up...
Part III

Proving Entailments
**Induction and Linked Lists**

List segments  \((\text{list}(E) \text{ is shorthand for } \text{lseg}(E, \text{nil}))\)

\[
\text{lseg}(E, F) \iff \begin{cases} 
E = F \text{ then } \text{emp} \\
\text{else } \exists y. E \mapsto \text{tl}: y \ast \text{lseg}(y, F)
\end{cases}
\]

\[
\text{lseg}(x, y) \ast \text{lseg}(y, x)
\]
Induction and Linked Lists

List segments  \((\text{list}(E)\text{ is shorthand for } \text{lseg}(E, \text{nil}))\)

\[
\text{lseg}(E, F) \iff \begin{cases} 
\text{if } E = F \text{ then emp} \\
\text{else } \exists y. E \mapsto tl : y \ast \text{lseg}(y, F)
\end{cases}
\]

\[
\text{lseg}(x, t) \ast t \mapsto [tl : y] \ast \text{list}(y)
\]
**Induction and Linked Lists**

List segments  \((\text{list}(E) \text{ is shorthand for } \text{lseg}(E, \text{nil}))\)

\[\text{lseg}(E, F) \iff \text{if } E = F \text{ then emp} \]

\[\text{else } \exists y. E \mapsto t\ell: y \ast \text{lseg}(y, F)\]

**Entailment**  \(\text{lseg}(x, t) \ast t\mapsto[t\ell: y] \ast \text{list}(y) \vdash \text{list}(x)\)
**Induction and Linked Lists**

List segments  
(list(E) is shorthand for lseg(E, nil))

\[ lseg(E, F) \iff \begin{cases} 
\text{if } E = F \text{ then } \text{emp} \\
\text{else } \exists y. E \mapsto \text{tl}: y \ast lseg(y, F) 
\end{cases} \]

Non-Entailment

\[ lseg(x, t) \ast t \mapsto \text{nil} \ast \text{list}(y) \not\models \text{list}(x) \]
Solution (Berdine and Calcagno)

- A proof theory oriented around Abstraction and Subtraction.
Solution (Berdine and Calcagno)

- A proof theory oriented around Abstraction and Subtraction.
- Sample Abstraction Rule

\[
\text{lseg}(x, t) \ast \text{list}(t) \vdash \text{list}(x)
\]
Solution (Berdine and Calcagno)

- A proof theory oriented around Abstraction and Subtraction.
- Sample Abstraction Rule

\[ \text{lseg}(x, t) \ast \text{list}(t) \vdash \text{list}(x) \]

- Subtraction Rule

\[
\frac{Q_1 \vdash Q_2}{Q_1 \ast S \vdash Q_2 \ast S}
\]
Solution (Berdine and Calcagno)

- A proof theory oriented around **Abstraction** and **Subtraction**.
- Sample Abstraction Rule

\[ \text{lseg}(x, t) \ast \text{list}(t) \vdash \text{list}(x) \]

- Subtraction Rule

\[
\frac{Q_1 \vdash Q_2}{Q_1 \ast S \vdash Q_2 \ast S}
\]

- Try to reduce an entailment to the axiom

\[
B \land \text{emp} \vdash \text{true} \land \text{emp}
\]
Works great!

\[ \text{lseg}(x, t) \ast t\mapsto [tl : y] \ast \text{list}(y) \vdash \text{list}(x) \]

Abstract (Roll)
Works great!

\[
\text{lseg}(x, t) \ast \text{list}(t) \vdash \text{list}(x) \\
lseg(x, t) \ast t \mapsto [tl : y] \ast \text{list}(y) \vdash \text{list}(x)
\]

Abstract (Inductive)
Abstract (Roll)
Works great!

\[ \text{list}(x) \vdash \text{list}(x) \]
\[ \text{lseg}(x, t) \times \text{list}(t) \vdash \text{list}(x) \]
\[ \text{lseg}(x, t) \times t \mapsto [tl : y] \times \text{list}(y) \vdash \text{list}(x) \]

Subtract
Abstract (Inductive)
Abstract (Roll)
Works great!

∓

emp ⊢ emp
list(x) ⊢ list(x)
lseg(x, t) * list(t) ⊢ list(x)
lseg(x, t) * t→[tl : y] * list(y) ⊢ list(x)

Axiom!
Subtract
Abstract (Inductive)
Abstract (Roll)
Works great!

Emp ⊢ Emp

list(x) ⊢ list(x)

lseg(x, t) * list(t) ⊢ list(x)

lseg(x, t) * t⇒[tl : y] * list(y) ⊢ list(x)

lseg(x, t) * t⇒nil * list(y) ⊢ list(x)

Axiom!

Subtract

Abstract (Inductive)

Abstract (Roll)

Abstract (Inductive)
Works great!

Axiom!

\[
\begin{align*}
\text{emp} & \vdash \text{emp} \\
\text{list}(x) & \vdash \text{list}(x) \\
\text{lseg}(x, t) \times \text{list}(t) & \vdash \text{list}(x) \\
\text{lseg}(x, t) \times t \mapsto [tl : y] \times \text{list}(y) & \vdash \text{list}(x) \\
\text{list}(x) \times \text{list}(y) & \vdash \text{list}(x) \\
\text{lseg}(x, t) \times t \mapsto \text{nil} \times \text{list}(y) & \vdash \text{list}(x)
\end{align*}
\]

Abstract (Inductive)

Abstract (Roll)

Subtract
Works great!

:-)

\[
\begin{align*}
\text{emp} & \vdash \text{emp} \\
\text{list}(x) & \vdash \text{list}(x) \\
\text{lseg}(x, t) \ast \text{list}(t) & \vdash \text{list}(x) \\
\text{lseg}(x, t) \ast t \mapsto [tl : y] \ast \text{list}(y) & \vdash \text{list}(x)
\end{align*}
\]

Axiom!

Subtract

Abstract (Inductive)

Abstract (Roll)


\[
\begin{align*}
\text{list}(y) & \vdash \text{emp} \\
\text{list}(x) \ast \text{list}(y) & \vdash \text{list}(x) \\
\text{lseg}(x, t) \ast t \mapsto \text{nil} \ast \text{list}(y) & \vdash \text{list}(x)
\end{align*}
\]

Junk: Not Axiom!

Subtract

Abstract (Inductive)
List of abstraction rules for $lseg$

**Rolling**

\[ emp \rightarrow lseg(E, E) \]

\[ E_1 \neq E_3 \land E_1 \mapsto [tl: E_2, \rho] \ast lseg(E_2, E_3) \rightarrow lseg(E_1, E_3) \]

**Induction Avoidance**

\[ lseg(E_1, E_2) \ast lseg(E_2, \text{nil}) \rightarrow lseg(E_1, \text{nil}) \]

\[ lseg(E_1, E_2) \ast E_2 \mapsto [t: \text{nil}] \rightarrow lseg(E_1, \text{nil}) \]

\[ lseg(E_1, E_2) \ast lseg(E_2, E_3) \ast E_3 \mapsto [\rho] \rightarrow lseg(E_1, E_3) \ast E_3 \mapsto [\rho] \]

\[ E_3 \neq E_4 \land lseg(E_1, E_2) \ast lseg(E_2, E_3) \ast lseg(E_3, E_4) \rightarrow lseg(E_1, E_3) \ast lseg(E_3, E_4) \]
Proof Procedure for $Q_1 \vdash Q_2$, Normalization Phase

» Substitute out all equalities

$$Q_1[E/x] \vdash Q_2[E/x]$$

\[ x = E \land Q_1 \vdash Q_2 \]

» Generate disequalities. E.g., using

$$x \mapsto [\rho] \ast y \mapsto [\rho'] \rightarrow x \neq y$$

» Remove empty lists and trees: \text{lseg}(x, x), \text{tree}(\text{nil})

» Check antecedent for inconsistency, if so, return “valid”.

Inconsistencies: $x \mapsto [\rho] \ast x \mapsto [\rho'] \quad \text{nil} \mapsto \neg \quad x \neq x \quad \cdots$

» Check pure consequences (easy inequational logic), if failed then “invalid”

This is cubic.
Proof Procedure for $Q_1 \vdash Q_2$, Abstract/Subtract Phase

Trying to prove $B_1 \land H_1 \vdash H_2$

- For each spatial predicate in $H_2$, try to apply abstraction rules to match it with things in $H_1$.
- Then, apply subtraction rule.

$$Q_1 \vdash Q_2$$

$$\frac{}{Q_1 \ast S \vdash Q_2 \ast S}$$

- If you are left with

$$B \land \text{emp} \vdash \text{true} \land \text{emp}$$

report “valid”, else “invalid”

This is cubic.
Completeness?

Question: from O’Hearn to Berdine/Calcago (circa 2002):

Is your procedure complete (and if not can you prove undecidability)?
Completeness?

- **Question**: from O’Hearn to Berdine/Calcago (circa 2002):
  
  *Is your procedure complete (and if not can you prove undecidability)?*

- **Immediate Answer**: silence
Completeness?

- **Question**: from O’Hearn to Berdine/Calcago (circa 2002): 
  
  *Is your procedure complete (and if not can you prove undecidability)?*

- **Immediate Answer**: silence

- **A little later**: Doh!
**Spooky Disjunctions**

- The fragment does not have disjunction in it. However,

\[ y \neq z \land (ls(x, y) \ast ls(x, z)) \]

implies that either \( ls(x, y) \) or \( ls(x, z) \) is nonempty, but we do not know which. So it also implies \( x \neq y \lor x \neq z \).
Spooky Disjunctions

- The fragment does not have disjunction in it. However,

\[ y \neq z \land (ls(x, y) \ast ls(x, z)) \]

implies that either \( ls(x, y) \) or \( ls(x, z) \) is nonempty, but we do not know which. So it also implies \( x \neq y \lor x \neq z \).

- This issue can show up in an entailment

\[ y \neq z \land (ls(x, y) \ast ls(x, z) \ast x \mapsto \neg) \vdash x \neq x \]

which tricks the proof procedure.
Spooky Disjunctions

The fragment does not have disjunction in it. However,

\[ y \neq z \land (ls(x, y) \ast ls(x, z)) \]

implies that either \( ls(x, y) \) or \( ls(x, z) \) is nonempty, but we do not know which. So it also implies \( x \neq y \lor x \neq z \).

This issue can show up in an entailment

\[ y \neq z \land (ls(x, y) \ast ls(x, z) \ast x \mapsto -) \vdash x \neq x \]

which tricks the proof procedure.

We have never fallen foul of this incompleteness in a natural example in program verification.
The fragment does not have disjunction in it. However,

\[ y \neq z \land (ls(x, y) \ast ls(x, z)) \]

implies that either \( ls(x, y) \) or \( ls(x, z) \) is nonempty, but we do not know which. So it also implies \( x \neq y \lor x \neq z \).

This issue can show up in an entailment

\[ y \neq z \land (ls(x, y) \ast ls(x, z) \ast x \mapsto \neg) \vdash x \neq x \]

which tricks the proof procedure.

We have never fallen foul of this incompleteness in a natural example in program verification.

Still...
Exorcising Spooky Disjunctions

- Cubic proof procedure is complete when we know all the listsegs are nonempty (when $x \neq y$ is there for each $\text{lseg}(x, y)$).
Exorcising Spooky Disjunctions

- Cubic proof procedure is complete when we know all the listsegs are nonempty (when $x \neq y$ is there for each $\text{lseg}(x, y)$).
- Complete procedure for general case, using excluded middle

$$
\frac{x = y \land Q_1 \vdash Q_2 \quad x \neq y \land Q_1 \vdash Q_2}{Q_1 \vdash Q_2}
$$
Cubic proof procedure is complete when we know all the listsegs are nonempty (when $x \neq y$ is there for each $\text{lseg}(x, y)$).

Complete procedure for general case, using excluded middle

$$
\begin{align*}
\frac{x = y \land Q_1 \vdash Q_2}{x \neq y \land Q_1 \vdash Q_2}
\end{align*}
$$

The resulting proof procedure is exponential.
Exorcising Spooky Disjunctions

- Cubic proof procedure is complete when we know all the listsegs are nonempty (when \( x \neq y \) is there for each \( \text{lseg}(x, y) \)).
- Complete procedure for general case, using excluded middle

\[
\begin{align*}
    x = y & \quad \text{\&} \quad Q_1 \vdash Q_2 \\
    x \neq y & \quad \text{\&} \quad Q_1 \vdash Q_2
\end{align*}
\]

\[
\Rightarrow \quad Q_1 \vdash Q_2
\]

- The resulting proof procedure is exponential.
- Calcagno has a polynomial procedure which uses constraints, and which handles spooky disjunctions. Don’t know if complete.
**Perspective**

- The interesting part is the abstraction rules replacing induction, like

\[
lseg(x, t) \ast \text{list}(t) \vdash \text{list}(x)
\]

- We can replay this work for other data structures, but (presently) some invention is needed to choose abstraction rules.
- We can infer loop invariants (abstract interpretation) by selective use of the abstraction rules.
Perspective

- The interesting part is the abstraction rules replacing induction, like

\[ \text{lseg}(x, t) \ast \text{list}(t) \vdash \text{list}(x) \]

- We can replay this work for other data structures, but (presently) some invention is needed to choose abstraction rules.
- We can infer loop invariants (abstract interpretation) by selective use of the abstraction rules.
  - Distefano et al, TACAS’06
Perspective

▶ The interesting part is the abstraction rules replacing induction, like

\[ \text{lseg}(x, t) \ast \text{list}(t) \vdash \text{list}(x) \]

▶ We can replay this work for other data structures, but (presently) some invention is needed to choose abstraction rules.

▶ We can infer loop invariants (abstract interpretation) by selective use of the abstraction rules.

▶ Distefano et al, TACAS’06

▶ Termination analysis: Berdine et al, CAV’06 (see Cook CAV invited)
  Interprocedural shape analysis: Gotsman et al, SAS’06
  Pointer Arithmetic: Calcagno et al, SAS’06
  Thread-modular shape analysis: Gotsman et al, PLDI’07
  Induction Synthesis: Guo et al, PLDI’07
  Adaptive analysis: Bertine et al, CAV’07 (see Distefano CAV talk)
  + 5 SAS’07 papers
Coalescing

- Freeing sometimes causes adjacent nodes to be coalesced in the free list.
- For example,

\[
[x+1] := [x+1] + [x+z+1]
\]
Coalescing

- Freeing sometimes causes adjacent nodes to be coalesced in the free list.
- For example,

\[
[x+1] := [x+1] + [x+z+1]
\]
Coalescing

- Freeing sometimes causes adjacent nodes to be coalesced in the free list.
- For example,

\[ \text{[x+1]} := \text{[x+1]} + \text{[x+z+1]} \]
RAM to Node transit

- Abstraction for transit from

\[
\begin{align*}
& y \rightarrow z + b \\
& a \rightarrow b \\
& x+1 \rightarrow x+2 \rightarrow x+z
\end{align*}
\]

is an implication

\[
(x \mapsto y) \ast (x+1 \mapsto z + b) \ast \text{blk}(x+2, x+z) \\
\ast (x+z \mapsto a) \ast (x+z+1 \mapsto b) \ast \text{blk}(x+z+2, x+z+b)
\]

\[
\implies nd(x, y, z + b)
\]
<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>Heap (KB)</th>
<th>States (Inv)</th>
<th>States (Post)</th>
<th>Time (sec)²</th>
</tr>
</thead>
<tbody>
<tr>
<td>malloc_firstfit</td>
<td>42</td>
<td>240</td>
<td>18</td>
<td>3</td>
<td>0.05</td>
</tr>
<tr>
<td>free_acyclic</td>
<td>55</td>
<td>240</td>
<td>6</td>
<td>2</td>
<td>0.09</td>
</tr>
<tr>
<td>malloc_besttfit</td>
<td>46</td>
<td>480</td>
<td>90</td>
<td>3</td>
<td>1.19</td>
</tr>
<tr>
<td>malloc_roving</td>
<td>61</td>
<td>240</td>
<td>33</td>
<td>5</td>
<td>0.13</td>
</tr>
<tr>
<td>free_roving</td>
<td>68</td>
<td>720</td>
<td>16</td>
<td>2</td>
<td>0.84</td>
</tr>
<tr>
<td>malloc_K&amp;R</td>
<td>179</td>
<td>26880</td>
<td>384</td>
<td>66</td>
<td>502.23</td>
</tr>
<tr>
<td>free_K&amp;R</td>
<td>58</td>
<td>3840</td>
<td>89</td>
<td>5</td>
<td>9.69</td>
</tr>
</tbody>
</table>

²Pentium 2.3GHz, 4GB RAM